07 Jan 1993

Intermediate Code Generation for Portable Scalable, Compilers. Architecture Independent Data Parallelism: The Preliminaries

Lenore Mullin

C. Chang

S. Huang

Matthias Mayer

et. al. For a complete list of authors, see https://scholarsmine.mst.edu/comsci_techreports/24

INTERMEDIATE CODE GENERATION FOR PORTABLE, SCALABLE, COMPILERS. ARCHITECTURE INDEPENDENT DATA PARALLELISM: THE PRELIMINARIES

CSC-93-01

L. Mullin, C. Chang, S. Huang, M. Mayer,
N. Nemer and C. Ramakrishna

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401

Intermediate Code Generation for Portable, Scalable, Compilers. Architecture Independent Data Parallelism: the preliminaries *

L. Mullin C. Chang S. Huang M. Mayer N. Nemer
C. Ramakrishna †

January 7, 1993

Abstract

This paper introduces the goals of the *P*ortable, *S*calable, Architecture *I*ndependent (*PSI*) Compiler Project for Data Parallel Languages at the University of Missouri-Rolla. A goal of this project is to produce a *subcompiler* for data parallel scientific programming languages such as HPF(High Performance Fortran) where the input grammar is translated to a three-address code intermediate language. Ultimately we plan to integrate our work into automated synthesis systems for scientific programming because we feel that it should not be necessary to learn complicated programming techniques to use multiprocessor computers or networks of computers effectively. This paper shows how to compile a data parallel language to an arbitrary multiprocessor topology or network of CPUs given the number of processors, length of vector registers, and total number of components in an array assuming a message passing, distributed memory paradigm of *send* and *receive*. We emphasize that this paradigm is not only amenable to machines such as the CM5 and NCube but to LAN and WAN connected architectures. We do *automatic* program partitioning and mapping to processing elements of

*This work was supported by the Natural Sciences and Engineering Council of Canada

†lenore@cs.umr.edu, cchange@cs.umr.edu, shuang@cs.umr.edu, matze@cs.umr.edu nnemer@cs.umr.edu, ramakr@cs.umr.edu Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri 65401

a multiprocessor architecture or distributed network of machines. No programmer intervention is required, hence, no errors will be introduced through data decomposition.
Keywords: data parallelism, supercompilers, data partitioning, scientific programming, intermediate langauges

1 Introduction

Over the past 10 years there has emerged a plethora of multiprocessing architectures. Initially, vector registers were added to uniprocessor architectures, e.g. IBM 3090. We later saw multiprocessing architectures with scalar and vector registers that used shared memory, e.g. Encore and Alliant respectively. These machines were basically scaled-up versions of uniprocessor architectures. Architectural and operating system designs were complicated by the fact that all processors needed to access shared memory as well as utilize their scalar/vector registers and multiple processors efficiently. Unfortunately, shared memory machines did not scale well. Hence, a more general scheme of *sharing data* needed to be devised that would scale as more processors were added. The distributed memory message passing paradigm is both general and scalable on an arbitrary topology of multiple CPUs using the *send* and *receive* primitives[1]. This paradigm is also applicable to many machines connected via a LAN or WAN[2].

Complexity is introduced when we wish to *localize* data to minimize communications costs[3]. This requires knowledge of the topology as well as the algorithm. Hence, it is difficult to find an architecture independent way to send and receive messages among processors and perform automatic partitioning of arrays, the key data structure in scientific programming. Large-scale parallelism is a viable approach to achieving high-performance computing. However, before these opportunities can be fully realized, many challenges and obstacles remain. In particular, software must scale and port easily as new computational platforms are introduced. The long term goals of this project are to build a *subcompiler* for existing *scientific* programming languages whose primary data structure is the array. An open question is how to *automatically* determine a suitable data partitioning scheme given an arbitrary number of processors using distributed memory machines. Existing methodologies require programmer intervention through compiler directives and/or explicit partitioning[4, 5]. Algorithmic correctness is compromised as soon as a formal design undergoes change via human intervention.

2 Subcompiler Front End

The efficient use of multiprocessing architectures is limited to highly skilled scientific parallel programmers, but scientists may want to run their own experiments by giving a high-level specification of an algorithm[6, 7] for wave propa gation or electromagnetics.

Preliminary work for our compiler takes as input a calculator language that can be augmented to include arrays as arguments. We assume that we can translate the grammar of any scientific language to the grammar of our calculator language. The basic idea is that scalars, vectors, or matrices, e.g., are actually zero, one, or two dimensional arrays[8]. If we can find a *generic* way to represent array operations, then a data parallel compiler that is transparent to users is possible. We have built all parts of the compiler, the lexical analyzer, symbol table, and parser. We employed various techniques for symbol table management and parsing. For symbol table management we implemented linear searches, binary searches and hash tables. Through comparative performance studies we determined that a hash table and deterministic bottom up push down automata parser(LR(1)) performed best. Being in an LR(k) class of parsers, it can be used on context free grammars, which is a powerful set of grammars. We also include removal of left recursion.

Intermediate code was then generated. We chose the three-address statement approach to resemble an abstract representation for intermediate code. The three-address statment method represents a linearized version of the DAG or syntax tree in which explicit names correspond to the interior nodes of the graph. A goal in any optimizing compiler is to minimize the creation of temporary variables. A classical way to do this is to represent the syntax tree as a DAG. We reused temporaries as often as possible.

We did not address techniques used to detect and exploit functional parallelism. We did however address how to develop a general *data parallelism* strategy by augmenting the design for *three address* intermediate code. Our designs will not only deal with scalar register allocation we will deal with vector register allocation and multiple processor allocation in a message passing environment.

The designs that follow will work on any message passing topology with or without vector registers. For now we will assume than an array is flattened to a vector(or one dimensional array) using a lexicographic ordering. This preliminary work provides the foundation to address multi-dimensional homogeneous arrays and will be the topic of a later paper.

Therefore our three address intermediate code will handle::

- scalars, uniprocessors, scalar registers

```
Temp = A + B;
```

Figure 1: Three Address Intermediate Code in a Classical Compiler

- vectors(or flattened arrays), uniprocessors, scalar registers
- vectors, uniprocessors, vector registers
- vectors, multiprocessors, scalar registers
- vectors, multiprocessors, vector registers.

We chose the C programming language as our intermediate code representation because we to be portable.[1] We also wanted a UNIX environment so that we could use sends and receives through socket connections for connecting processing elements through TCP/IP.

3 Intermediate Code Generation

Prior to intermediate code generation in a classical compiler, optimizations are first performed performed to minimize expression trees and reuse temporaries. The three address intermediate code in Figure 1 puts Temp, A, and B into registers and is followed by a register to register addition. Unfortunately each argument of the above expression denotes a scalar operations using scalar registers. But what if we could augment the grammar to include arrays[2]. We would want to have a three address code representation that put arrays into scalar or vector registers over one or more processors.

3.0.1 Generic Three Address Intermediate Code for Multiprocessors and Vector Registers

Scalars are vectors of length zero and matrices are vectors whose components are vectors. Three dimensional arrays (cubes) are vectors whose components are matrices. Vectors are one dimensional arrays. Therefore, we can represent an n-dimensional array (n=0,1,2,...) with a one dimensional array by flattening its components.

[1] A general scheme requires that we can easily manipulate addresses

[2] When we, for now, talk about performing scalar operation between two arrays, such as plus, minus, times, or divide, we mean a componentwise operations. For higher order operations we plan to incorporate the ψ calculus [9]

Consider a vector expression[3]

```
Temp = A + B;
```

in which A, B, and Temp are vectors of the same length N. In a sequential program we would perform

```
for i = 1 to N do
     Temp[i] = A[i] + B[i];
```

In a vector machine with vector registers of length L, it is:

```
for i = 1 to N step L do
{the following is a vector operation}
for j = (i-1)*L to min(i*L,N) do
     Temp[j] = A[j] + B[j];
```

Suppose that we have P processors, we can distribute all the components of vectors to p processors uniformly:

```
{processor loop}
for p = 1 to P do
  for i = 1 to N/P step L do
      {vector operation}
        for j = (p-1)*N/P+(i-1)*L to min ((p-1)*N/P+1*L, p*N/P) do
        Temp[j] = A[j] + B[j];
```

What we are trying to illustrate is how to develop a prototype of array operations for multiple processors and vector registers. But, we also want the end user to be able to simply say add array A to array B, i.e.

$$\text{Temp} = \text{A} + \text{B}$$

given that A, B, and Temp are conformable for a pointwise addition. With this in mind, we can generate intermediate code in the programming language C to represent array operations. The following is a simple example which calculates the above expression.

The values of P, N, and L can be adjusted to fit the architecture and operations[4]

[3]Here we treat an array monolithically[10].

[4]See the Appendix for actual input and intermediate code output of our compiler.

```
    /* P is the number of processor available */
    /* N is the length of the flattened array */
    /* L is the length of the vector register */
    for (p=0; p<P; p++){      /* processor loop - the partitioning */
       if ((id = fork())<0)   /* we use fork to simulate multiple  */
                              /* on a uniprocessor                 */
           printf(''error'');
       else if (id == 0 )
         for (i=0, i< N/P; i++){ /* component loop - of the partition  */
           x= p * (N/P) +i;
           for ( j=0, j<L; j++) { /* vector register loop /*
               *(Temp+x+j) = *(A+x+j) + *(B+x+j);
           }  /* end of vector register loop */
         }  /* end of component loop */
       }  /* end of processor loop */
/* We will use sends and receives to pass messages eventually. */
/* The parent process will collect all results                 */
/* after all of its children are done.                         */
```

Figure 2: Generic Three Address Intermediate Code for Decomposition and Mapping to Multiproccssors and Vector Registers

4 Experimental results

As mentioned, we used a grammar that supports scalar(or 0-dimensional arrays) operations and data. For our experiment P, N, and L are all equal to one, because it is the special case of vector operations[8] and it is the special case for number of processors, i.e. a uniprocessor. Figure 3 illustrate the grammar for our calculator language.

```
<start> -> <idlist> ; program<stmtlist>;
<idlist> -> id | <idlist>, id
<stmtlist> -> <stmt> | <stmtlist> ; <stmt>
<stmt> -> id = <exp> | print<idlist>
<exp> -> <exp> + <term> | <exp> - <term> | <term>
<term> -> <term> * <factor> | <term> / <factor> | <factor>
<factor -> id | number | (exp)
```

Figure 3: Grammar for our scalar calculator language

As expected, our generic code did not perform as well as the classical scalar code[5]. This is because we were using scalars, scalar operations, and a uniprocessor. For scalar operations the generic loops are not necessary. Our purpose was not to generate the best scalar code but to generate generic code for array operations(where scalars are 0 dimensional arrays) that could be partitioned and routed to a message passing multiprocessing architecture(s). If the lengths of the vectors(or flattened arrays) are significantly large than the performance should be better due to parallelization and vectorization. This will also be the topic of a subsequent paper.

5 Conclusion and Future Research

We have introduced the goals of the *PSI* compiler project at UMR. Our scalar calculator language now supports a generic three-address code for multiple processors and vector registers. Output of different parsers, LL(1), LR(1) and LALR(1) produced the same results after running through the C compiler. We feel that the three-address intermediate code using a subset of C gives us a lot of versatility since C supports pointers. The foundations now exist to augment our compiler to include multidimensional arrays partitioned over multiple processors, workstations, and/or vector registers.

We plan to add arrays and an associated algebra to our calculator language. After we have support for arrays we will then target the CM5, NCUBE and a group of TCP/IP connected workstations. It is our hope that the promise of portable, scalable efficient support of data parallel operations will be realized. We also plan to investigate how we would include this research in a subcompiler for HPF as well as an automated synthesis system.

References

[1] N. Bélanger, L. Mullin, and Y. Savaria. Formal methods for the partitioning, scheduling and routing of arrays on a hierarchical bus multiprocessing architecture. Technical Report CSEE/92/06-04, University of Vermont, Dept of CSEE, 1992. Presented at ATABLE92, Montréal Québec, 1992, TR No. 841, University of Montreal.

[5] See the Appendix for actual input and output of our compiler.

[2] L. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for the scheduling, routing and communications protocol of a logarithmic scan on a message passing distributed operating system: Pram algorithms revisited. Technical Report CSEE/92/07-10, University of Vermont, Dept of CSEE, 1992. Under Review, IEEE Transactions on Parallel and Distributed Computing.

[3] S. Thibault. An automatic parallelizing compiler for operations on arrays. Technical Report CSEE/92/12-14, University of Vermont, Department of CS and EE, 1992.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. *A Users Guide to PVM, Parallel Virtual Machine.* Oak Ridge National Laboratory. Report ORNL/TM-11826.

[5] V.S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency Practice and Experience*, 1990.

[6] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Automated Synthesis of Finite Difference Programs. In *Symbolic Computations and Their Impact on Mechanics, PVP-Volume 205.* The American Society of Mechanical Engineers 1990, New York, NY, 1990. ISBN 0-791800598-0.

[7] Elaine Kant, François Daube, William MacGregor, and Joseph Wald. Scientific Programming by Automated Synthesis. In M. Lowry and R. McCartney, editors, *Automating Software Design.* AAAI Press, 1991.

[8] R. Allen and K. Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10), 1992.

[9] Lenore M. Restifo Mullin. *A Mathematics of Arrays.* Ph.D. dissertation, Syracuse University, December 1988.

[10] G. Gao, R. Yates, L. Mullin, and J. Dennis. An efficient monolithic array constructor for scientific computation. In *Proceedings of the Third Workshop on Programming Languages and Compilers of Parallel Computing.* MIT Press, 1990.

Appendix

Input to our Compiler

```
var
  FUX8LpWnT8,
  LhVGj7zEGO,
  c1CQrJJVCR,
  wNliMxLgac,
  C59x9ivxmz,
  VaoCoVLypN,
  NzbpVlagFy,
  Ris6VpfGcF,
  deeieI3p4i,
  N8WQnrzBO4,
  fLafHwLk0n,
  suFISjQJNL,
  SCHESstFt5,
  DgGYAMKLu8,
  wYs0gAZvKk,
  w0m922rl5a,
  X34lR4vZDg,
  l02GqZEDIM,
  SAAM90Fqc8,
  y716gG1Sf2;

program
  FUX8LpWnT8=4254*(2369)/0152+(6878-4612*5593/3472)+(8651)-5393;
  LhVGj7zEGO=7807*3386/(3627+0404)-9178*(3346/4771+4986)-0994;
  c1CQrJJVCR=(FUX8LpWnT8)*(5796/4491+FUX8LpWnT8)-(FUX8LpWnT8*8356);
  wNliMxLgac=FUX8LpWnT8/(2921+8641)-FUX8LpWnT8*LhVGj7zEGO/8751;
  C59x9ivxmz=(5657+0399)-0844*(4913)/5757+LhVGj7zEGO-7157;
  VaoCoVLypN=(6391*4648)/(7472)+4993-7755*5798/(6461)+8198;
  NzbpVlagFy=FUX8LpWnT8-(9989*9576)/(8144)+LhVGj7zEGO-5323;
  Ris6VpfGcF=(2923)*(2115)/6354+2614-(FUX8LpWnT8*VaoCoVLypN)/4295;
  deeieI3p4i=(6644)+(8934-8077)*(1991/7575+8522-6861*8188);
```

```
N8WQnrzBO4=C59x9ivxmz/(5062)+(4598-3259*1318/8617)+3875;
fLafHwLk0n=deeieI3p4i-(3293)*6834/(2426)+6704-8293*2288;
suFISjQJNL=7537/(1786+1065-5895)*5379/(0215)+(1509)-1354;
SCHESstFt5=(N8WQnrzBO4*3767/2566+2848-5287)*(LhVGj7zEGO)/6552;
DgGYAMKLu8=7364+c1CQrJJVCR-(N8WQnrzBO4)*1161/(3197+8933);
wYs0gAZvKk=(7292)-(7804*LhVGj7zEGO)/(2918+9032-6264)*2114;
w0m922rl5a=(2998)/(2791)+(5441-6408)*FUX8LpWnT8/(8127+1479);
X34lR4vZDg=c1CQrJJVCR-(DgGYAMKLu8*Ris6VpfGcF/3204)+7719;
l02GqZEDIM=c1CQrJJVCR-5422*(6004/2648+3622)-3853*5373/6532;
SAAM90Fqc8=(0243)+(VaoCoVLypN-SCHESstFt5*3319/0589)+5384;
y716gG1Sf2=deeieI3p4i-7513*(fLafHwLk0n/8241)+(8985-1389);
print FUX8LpWnT8;
print LhVGj7zEGO;
print c1CQrJJVCR;
print wNliMxLgac;
print C59x9ivxmz;
print VaoCoVLypN;
print NzbpVlagFy;
print Ris6VpfGcF;
print deeieI3p4i;
print N8WQnrzBO4;
print fLafHwLk0n;
print suFISjQJNL;
print SCHESstFt5;
print DgGYAMKLu8;
print wYs0gAZvKk;
print w0m922rl5a;
print X34lR4vZDg;
print l02GqZEDIM;
print SAAM90Fqc8;
print y716gG1Sf2;
```

Generic Three-Address Intermediate Code
for Multiple Processors and Vector Registers

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>
struct rusage sptr,eptr;
 float tmp[20];
 void genfunct(char,float *,float *,float *);
main()
 {
float
 FUX8LpWnT8, LhVGj7zEGO, c1CQrJJVCR, wNliMxLgac, C59x9ivxmz, VaoCoVLypN,
 NzbpVlagFy, Ris6VpfGcF, deeieI3p4i, N8WQnrzBO4, fLafHwLkOn, suFISjQJNL,
 SCHESstFt5, DgGYAMKLu8, wYsOgAZvKk, wOm922rl5a, X34lR4vZDg, l02GqZEDIM,
 SAAM90Fqc8, y716gG1Sf2;
unsigned long t1,t2;
int i;
getrusage(RUSAGE_SELF,&sptr);
for(i=0;i<2000;i++){
tmp[1] = 4254;
 tmp[2] = 2369;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 152;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
tmp[6] = 6878;
 tmp[7] = 4612;
 tmp[8] = 5593;
    genfunct('*',&tmp[9],&tmp[7],&tmp[8]);
tmp[10] = 3472;
    genfunct('/',&tmp[11],&tmp[9],&tmp[10]);
  genfunct('-',&tmp[12],&tmp[6],&tmp[11]);
  genfunct('+',&tmp[13],&tmp[5],&tmp[12]);
tmp[14] = 8651;
   genfunct('+',&tmp[15],&tmp[13],&tmp[14]);
```

```
tmp[16] = 5393;
   genfunct('-',&tmp[17],&tmp[15],&tmp[16]);
   genfunct(' ',&FUX8LpWnT8,&tmp[17],&tmp[17]);
tmp[1] = 7807;
 tmp[2] = 3386;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 3627;
 tmp[5] = 404;
   genfunct('+',&tmp[6],&tmp[4],&tmp[5]);
   genfunct('/',&tmp[7],&tmp[3],&tmp[6]);
tmp[8] = 9178;
 tmp[9] = 3346;
 tmp[10] = 4771;
    genfunct('/',&tmp[11],&tmp[9],&tmp[10]);
tmp[12] = 4986;
   genfunct('+',&tmp[13],&tmp[11],&tmp[12]);
   genfunct('*',&tmp[14],&tmp[8],&tmp[13]);
  genfunct('-',&tmp[15],&tmp[7],&tmp[14]);
tmp[16] = 994;
   genfunct('-',&tmp[17],&tmp[15],&tmp[16]);
   genfunct(' ',&LhVGj7zEGO,&tmp[17],&tmp[17]);
tmp[1] = 5796;
 tmp[2] = 4491;
    genfunct('/',&tmp[3],&tmp[1],&tmp[2]);
  genfunct('+',&tmp[4],&tmp[3],&FUX8LpWnT8);
   genfunct('*',&tmp[5],&FUX8LpWnT8,&tmp[4]);
tmp[6] = 8356;
    genfunct('*',&tmp[7],&FUX8LpWnT8,&tmp[6]);
  genfunct('-',&tmp[8],&tmp[5],&tmp[7]);
   genfunct(' ',&c1CQrJJVCR,&tmp[8],&tmp[8]);
tmp[1] = 2921;
 tmp[2] = 8641;
   genfunct('+',&tmp[3],&tmp[1],&tmp[2]);
   genfunct('/',&tmp[4],&FUX8LpWnT8,&tmp[3]);
```

```
   genfunct('*',&tmp[5],&FUX8LpWnT8,&LhVGj7zEGO);
tmp[6] = 8751;
    genfunct('/',&tmp[7],&tmp[5],&tmp[6]);
  genfunct('-',&tmp[8],&tmp[4],&tmp[7]);
   genfunct(' ',&wNliMxLgac,&tmp[8],&tmp[8]);
tmp[1] = 5657;
 tmp[2] = 399;
   genfunct('+',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 844;
 tmp[5] = 4913;
    genfunct('*',&tmp[6],&tmp[4],&tmp[5]);
tmp[7] = 5757;
    genfunct('/',&tmp[8],&tmp[6],&tmp[7]);
  genfunct('-',&tmp[9],&tmp[3],&tmp[8]);
  genfunct('+',&tmp[10],&tmp[9],&LhVGj7zEGO);
tmp[11] = 7157;
   genfunct('-',&tmp[12],&tmp[10],&tmp[11]);
   genfunct(' ',&C59x9ivxmz,&tmp[12],&tmp[12]);
tmp[1] = 6391;
 tmp[2] = 4648;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 7472;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
tmp[6] = 4993;
   genfunct('+',&tmp[7],&tmp[5],&tmp[6]);
tmp[8] = 7755;
 tmp[9] = 5798;
    genfunct('*',&tmp[10],&tmp[8],&tmp[9]);
tmp[11] = 6461;
    genfunct('/',&tmp[12],&tmp[10],&tmp[11]);
  genfunct('-',&tmp[13],&tmp[7],&tmp[12]);
tmp[14] = 8198;
   genfunct('+',&tmp[15],&tmp[13],&tmp[14]);
   genfunct(' ',&VaoCoVLypN,&tmp[15],&tmp[15]);
```

```
tmp[1] = 9989;
 tmp[2] = 9576;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 8144;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
  genfunct('-',&tmp[6],&FUX8LpWnT8,&tmp[5]);
  genfunct('+',&tmp[7],&tmp[6],&LhVGj7zEGO);
tmp[8] = 5323;
   genfunct('-',&tmp[9],&tmp[7],&tmp[8]);
   genfunct(' ',&NzbpVlagFy,&tmp[9],&tmp[9]);
tmp[1] = 2923;
 tmp[2] = 2115;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 6354;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
tmp[6] = 2614;
   genfunct('+',&tmp[7],&tmp[5],&tmp[6]);
   genfunct('*',&tmp[8],&FUX8LpWnT8,&VaoCoVLypN);
tmp[9] = 4295;
    genfunct('/',&tmp[10],&tmp[8],&tmp[9]);
  genfunct('-',&tmp[11],&tmp[7],&tmp[10]);
   genfunct(' ',&Ris6VpfGcF,&tmp[11],&tmp[11]);
tmp[1] = 6644;
 tmp[2] = 8934;
 tmp[3] = 8077;
   genfunct('-',&tmp[4],&tmp[2],&tmp[3]);
tmp[5] = 1991;
 tmp[6] = 7575;
    genfunct('/',&tmp[7],&tmp[5],&tmp[6]);
tmp[8] = 8522;
   genfunct('+',&tmp[9],&tmp[7],&tmp[8]);
tmp[10] = 6861;
 tmp[11] = 8188;
    genfunct('*',&tmp[12],&tmp[10],&tmp[11]);
```

```
  genfunct('-',&tmp[13],&tmp[9],&tmp[12]);
   genfunct('*',&tmp[14],&tmp[4],&tmp[13]);
  genfunct('+',&tmp[15],&tmp[1],&tmp[14]);
   genfunct(' ',&deeieI3p4i,&tmp[15],&tmp[15]);
tmp[1] = 5062;
    genfunct('/',&tmp[2],&C59x9ivxmz,&tmp[1]);
tmp[3] = 4598;
 tmp[4] = 3259;
 tmp[5] = 1318;
    genfunct('*',&tmp[6],&tmp[4],&tmp[5]);
tmp[7] = 8617;
    genfunct('/',&tmp[8],&tmp[6],&tmp[7]);
  genfunct('-',&tmp[9],&tmp[3],&tmp[8]);
  genfunct('+',&tmp[10],&tmp[2],&tmp[9]);
tmp[11] = 3875;
   genfunct('+',&tmp[12],&tmp[10],&tmp[11]);
   genfunct(' ',&N8WQnrzBO4,&tmp[12],&tmp[12]);
tmp[1] = 3293;
 tmp[2] = 6834;
    genfunct('*',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 2426;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
  genfunct('-',&tmp[6],&deeieI3p4i,&tmp[5]);
tmp[7] = 6704;
   genfunct('+',&tmp[8],&tmp[6],&tmp[7]);
tmp[9] = 8293;
 tmp[10] = 2288;
    genfunct('*',&tmp[11],&tmp[9],&tmp[10]);
  genfunct('-',&tmp[12],&tmp[8],&tmp[11]);
   genfunct(' ',&fLafHwLk0n,&tmp[12],&tmp[12]);
tmp[1] = 7537;
 tmp[2] = 1786;
 tmp[3] = 1065;
   genfunct('+',&tmp[4],&tmp[2],&tmp[3]);
```

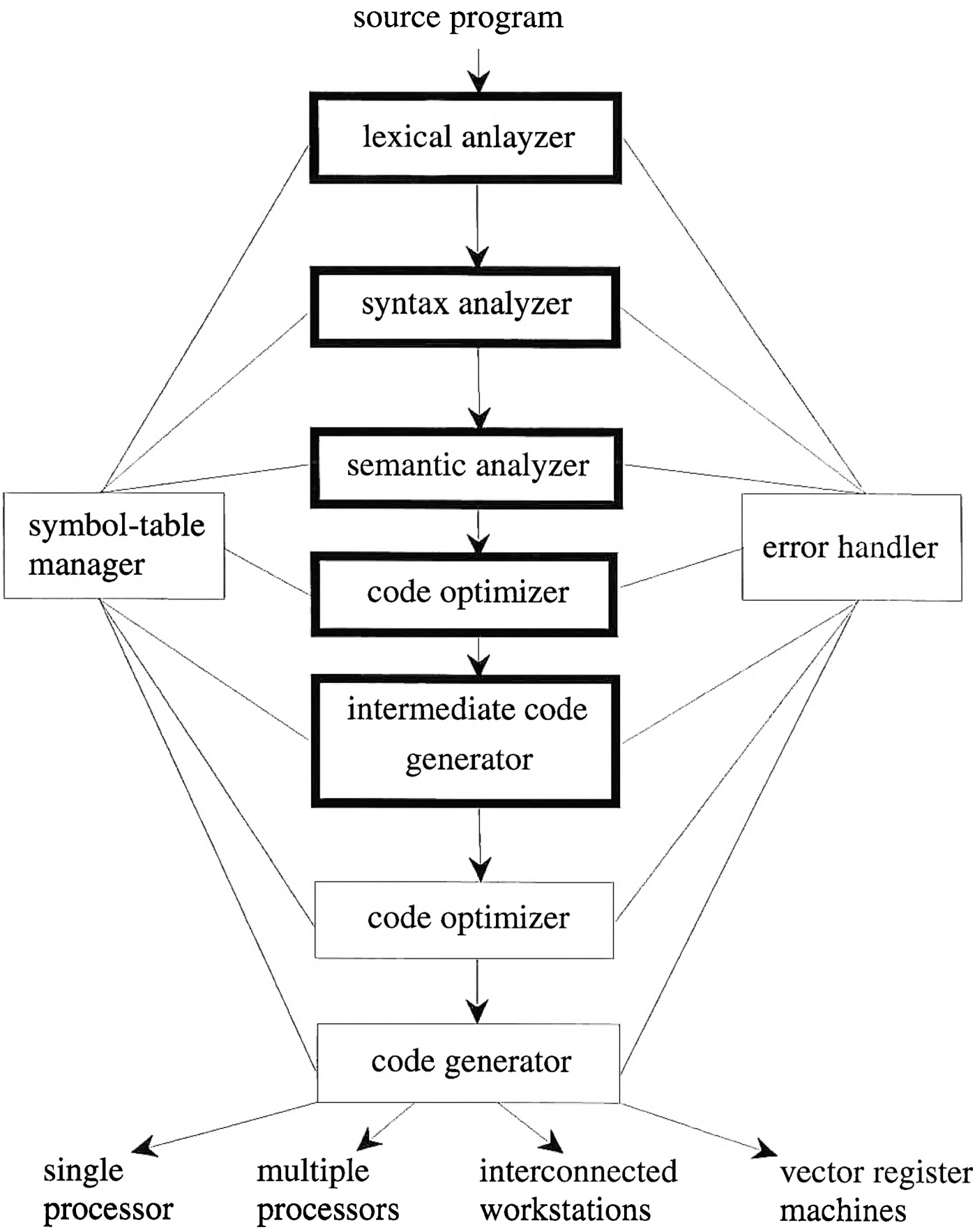
```
tmp[5] = 5895;
   genfunct('-',&tmp[6],&tmp[4],&tmp[5]);
   genfunct('/',&tmp[7],&tmp[1],&tmp[6]);
tmp[8] = 5379;
    genfunct('*',&tmp[9],&tmp[7],&tmp[8]);
tmp[10] = 215;
    genfunct('/',&tmp[11],&tmp[9],&tmp[10]);
tmp[12] = 1509;
   genfunct('+',&tmp[13],&tmp[11],&tmp[12]);
tmp[14] = 1354;
   genfunct('-',&tmp[15],&tmp[13],&tmp[14]);
   genfunct(' ',&suFISjQJNL,&tmp[15],&tmp[15]);
tmp[1] = 3767;
    genfunct('*',&tmp[2],&N8WQnrzBO4,&tmp[1]);
tmp[3] = 2566;
    genfunct('/',&tmp[4],&tmp[2],&tmp[3]);
tmp[5] = 2848;
   genfunct('+',&tmp[6],&tmp[4],&tmp[5]);
tmp[7] = 5287;
   genfunct('-',&tmp[8],&tmp[6],&tmp[7]);
   genfunct('*',&tmp[9],&tmp[8],&LhVGj7zEGO);
tmp[10] = 6552;
    genfunct('/',&tmp[11],&tmp[9],&tmp[10]);
   genfunct(' ',&SCHESstFt5,&tmp[11],&tmp[11]);
tmp[1] = 7364;
   genfunct('+',&tmp[2],&tmp[1],&c1CQrJJVCR);
tmp[3] = 1161;
    genfunct('*',&tmp[4],&N8WQnrzBO4,&tmp[3]);
tmp[5] = 3197;
 tmp[6] = 8933;
   genfunct('+',&tmp[7],&tmp[5],&tmp[6]);
   genfunct('/',&tmp[8],&tmp[4],&tmp[7]);
  genfunct('-',&tmp[9],&tmp[2],&tmp[8]);
   genfunct(' ',&DgGYAMKLu8,&tmp[9],&tmp[9]);
```

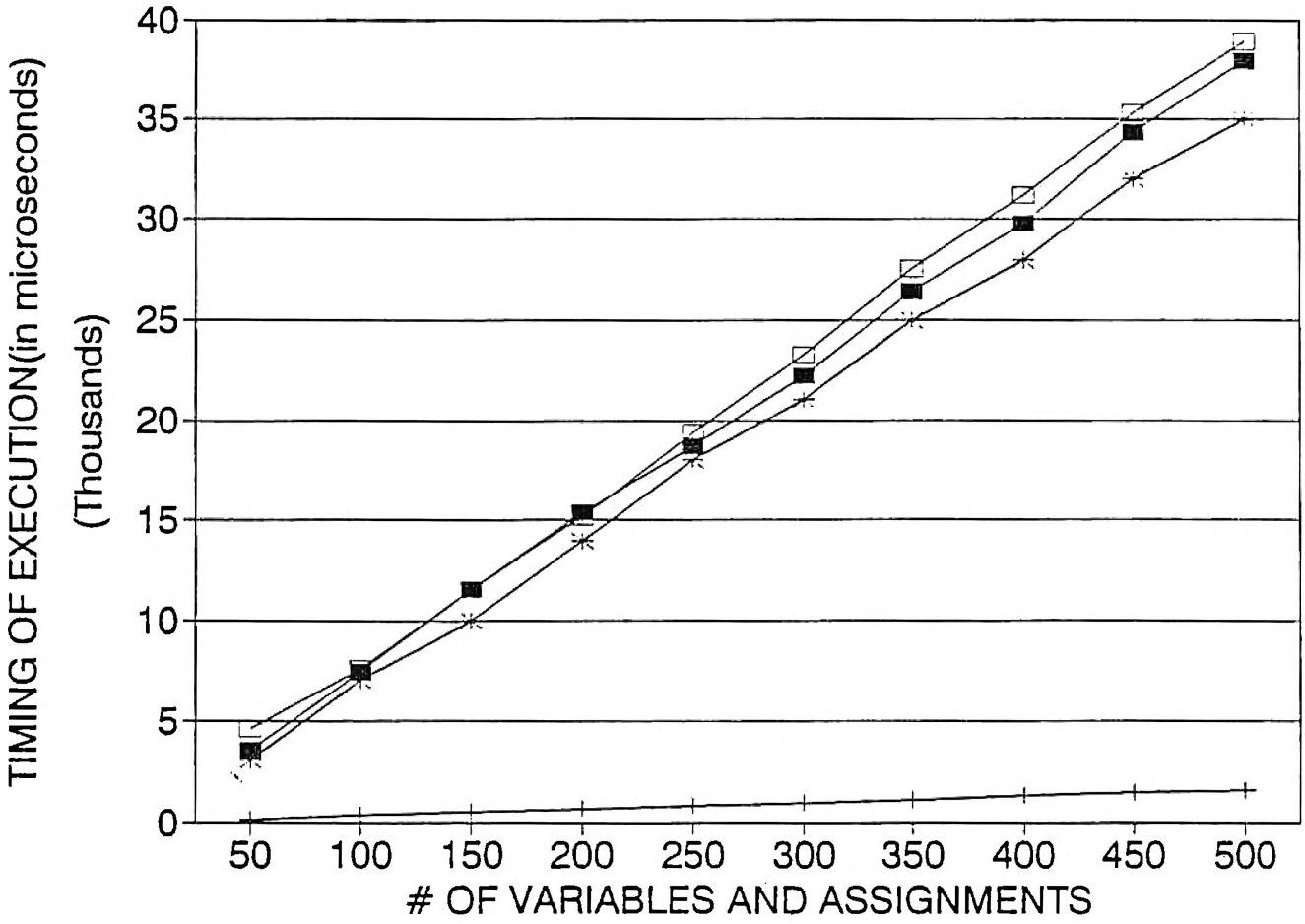
```
tmp[1] = 7292;
 tmp[2] = 7804;
    genfunct('*',&tmp[3],&tmp[2],&LhVGj7zEG0);
tmp[4] = 2918;
 tmp[5] = 9032;
   genfunct('+',&tmp[6],&tmp[4],&tmp[5]);
tmp[7] = 6264;
   genfunct('-',&tmp[8],&tmp[6],&tmp[7]);
   genfunct('/',&tmp[9],&tmp[3],&tmp[8]);
tmp[10] = 2114;
    genfunct('*',&tmp[11],&tmp[9],&tmp[10]);
  genfunct('-',&tmp[12],&tmp[1],&tmp[11]);
   genfunct(' ',&wYs0gAZvKk,&tmp[12],&tmp[12]);
tmp[1] = 2998;
 tmp[2] = 2791;
    genfunct('/',&tmp[3],&tmp[1],&tmp[2]);
tmp[4] = 5441;
 tmp[5] = 6408;
   genfunct('-',&tmp[6],&tmp[4],&tmp[5]);
   genfunct('*',&tmp[7],&tmp[6],&FUX8LpWnT8);
tmp[8] = 8127;
 tmp[9] = 1479;
   genfunct('+',&tmp[10],&tmp[8],&tmp[9]);
   genfunct('/',&tmp[11],&tmp[7],&tmp[10]);
  genfunct('+',&tmp[12],&tmp[3],&tmp[11]);
   genfunct(' ',&w0m922rl5a,&tmp[12],&tmp[12]);
   genfunct('*',&tmp[1],&DgGYAMKLu8,&Ris6VpfGcF);
tmp[2] = 3204;
    genfunct('/',&tmp[3],&tmp[1],&tmp[2]);
  genfunct('-',&tmp[4],&c1CQrJJVCR,&tmp[3]);
tmp[5] = 7719;
   genfunct('+',&tmp[6],&tmp[4],&tmp[5]);
   genfunct(' ',&X34lR4vZDg,&tmp[6],&tmp[6]);
tmp[1] = 5422;
```

```
 tmp[2] = 6004;
 tmp[3] = 2648;
    genfunct('/',&tmp[4],&tmp[2],&tmp[3]);
tmp[5] = 3622;
   genfunct('+',&tmp[6],&tmp[4],&tmp[5]);
   genfunct('*',&tmp[7],&tmp[1],&tmp[6]);
  genfunct('-',&tmp[8],&c1CQrJJVCR,&tmp[7]);
tmp[9] = 3853;
 tmp[10] = 5373;
    genfunct('*',&tmp[11],&tmp[9],&tmp[10]);
tmp[12] = 6532;
    genfunct('/',&tmp[13],&tmp[11],&tmp[12]);
  genfunct('-',&tmp[14],&tmp[8],&tmp[13]);
   genfunct(' ',&l02GqZEDIM,&tmp[14],&tmp[14]);
tmp[1] = 243;
 tmp[2] = 3319;
    genfunct('*',&tmp[3],&SCHESstFt5,&tmp[2]);
tmp[4] = 589;
    genfunct('/',&tmp[5],&tmp[3],&tmp[4]);
  genfunct('-',&tmp[6],&VaoCoVLypN,&tmp[5]);
  genfunct('+',&tmp[7],&tmp[1],&tmp[6]);
tmp[8] = 5384;
   genfunct('+',&tmp[9],&tmp[7],&tmp[8]);
   genfunct(' ',&SAAM90Fqc8,&tmp[9],&tmp[9]);
tmp[1] = 7513;
 tmp[2] = 8241;
    genfunct('/',&tmp[3],&fLafHwLk0n,&tmp[2]);
   genfunct('*',&tmp[4],&tmp[1],&tmp[3]);
  genfunct('-',&tmp[5],&deeieI3p4i,&tmp[4]);
tmp[6] = 8985;
 tmp[7] = 1389;
   genfunct('-',&tmp[8],&tmp[6],&tmp[7]);
  genfunct('+',&tmp[9],&tmp[5],&tmp[8]);
   genfunct(' ',&y716gG1Sf2,&tmp[9],&tmp[9]);
```

```
}
getrusage(RUSAGE_SELF,&eptr);
t1 = ( unsigned long)sptr.ru_utime.tv_sec * 1000000UL + (unsigned long)sptr.ru_utime.tv_u
t2 = (unsigned long)eptr.ru_utime.tv_sec * 1000000UL + (unsigned long)eptr.ru_utime.tv_us
printf("Total time : %d\n",(t2-t1)/2000);
}


 void genfunct( char op, float *result, float *left, float * right)
{


int K,I,X,H;
int T=1,P=1,L=1;
 int ID = 0;
for(H=0;H<P;H++)
{
if(ID == 0)
{
for(K=0;K<T/P;K++)
{
X = (H * (T/P)) + K;
switch(op)
{
case '+':
        for(I = 0;I < L;I++)
        *(result+X+I) = *(left + X + I) + *(right+X+I);
        break;

case '-' :
        for(I = 0;I< L;I++)
        *(result+X+I) = *(left + X + I) - *(right + X + I);
        break;
```

```
case '*' :
         for(I=0;I<L;I++)
         *(result+X+I) = *(left + X+ I) *  *(right +X + I);
         break;

case '/' :
        for(I=0;I<L;I++)
         *(result + X + I) = *(left + X + I) / *(right + X + I);
         break;

default :
        for(I=0;I<L;I++)
        *(result + X + I) = *(left + X + I) ;

 }
}
}
}
}
```


source program
lexical anlayzer
syntax analyzer
semantic analyzer
symbol-table manager
error handler
code optimizer
intermediate code generator
code optimizer
code generator
single processor
multiple processors
interconnected workstations
vector register machines



PERFORMANCE OF EXECUTABLE CODE USING GENERIC
THREE-ADDRESS CODE INTERMEDIATE CODE